Introduction

Problem Solving on Computer

Data Structures
(collection of data and relationships)

Algorithms
Objective of Data Structures

Two Goals:

1) Identify and develop useful high-level data types and operations and to determine what class of problems can be solved using them.

2) Implementation of these tools using already existing data types (such as arrays, lists, etc.)
Problem Solving Paradigm

Analyze the Problem
Use known Algorithms and Data Structure

Abstraction
(Space/Efficiency: Space-Time Trade-Off)
Select an appropriate Data Structure
Design Algorithm for Data Structure

Implementation
Translate into Target Language
Problem Solving Paradigm (Phases)

- Specify Requirements
- Designing Algorithm
- Refinement and Coding
- Verification
- Analyzing Algorithm
1. Specifying Requirements

- Understanding the given information (input), and what results to be produced.

- Writing a rigorous description of all the possible cases of inputs and outputs.
2. Designing Algorithm

Can the problem be solved algorithmically?

- Identifying operations to be performed on several data objects.
- Informally writing techniques or steps involved in processing (correctness??)
- Modularity (structural programming) is useful in verifying correctness.
- Simplicity (may not be efficient) is desirable. Good for writing, verifying correctness and debugging.
3. Analyzing Algorithm

Is the problem solvable practically?

Example: Chess Playing Program:

Consider all possible moves
(at least $10^9$ by some estimates)

A program which makes a thorough examination of them would take several thousand years to execute.
Considerations for Practical Importance (Time and Space Requirements (complexity)):

- How can a given algorithms and programs be improved?
- How can the efficiency of algorithms be analyzed?
- What criteria should be used to choose between different algorithms for the same application?
- In what sense can algorithms be shown to be the best possible?
Space-Time Trade-off

Trade-off
- amount of time required to perform algorithm versus
- amount of space

• Usually space constraints are less important.
4. Refinement and Coding

• Selection of a Data Structure
• Selection of Language (host consideration)

Suggestive approach is that all processing, which is independent of data structure must be written out first.
5. Verification

• Correctness or Program Proving

*Using Theorems and Lemmas*

Example: Validity of the Gauss elimination method for solving of linear equations, depends upon a number of theorems in linear algebra.

*Using Mathematical Induction*

• Testing
  - Creating sample data and run the program

• Debugging
Abstract Data Type

A method (concept) for interpreting the memory contents of a computer (which are bits)

Example: contents may represent integers, real numbers, characters, words, etc.

1) Logical properties, and,
2) Legal operations

need to be specified for proper interpretation.
Abstract Data Type

- Declaration
- Operations
- Implementation

A Semiformal Approach

1. **Value Definition**
   abstract type
   condition (optional)

2. **Operator(s) Definition(s)**
   abstract function
   pre-condition (optional)
   post-condition
   (specifies what the operation does)
What Basic Data Structure(s) Should be used to Implement Higher Level ADT?

- Arrays (static)
- Pointers/Liked List (dynamic)
Example: Array as an ADT

1-D Case:

abstract typedef<<eletype, ub>>
ARRTYPE (ub, eltype);
condition type(ub) == int,

Example of ARRTYPE:
int a[10];
Possible Functions:

• Get a value (Extraction via index, e.g. \( x=a[i] \) )
• Store a value (assignment statement \( a[3]=x \) )

ADT Representation: Internally as linear chain of memory.
**Implementation of Arrays (Logical to Physical Mapping)**

```c
int b[10];
```

- Allocate 10 successive memory locations, where the **base address** is the first location.
- Each element is of a particular size (memory units having a unique address), \( esize \).

| \( esize \) | \( b[0] \) | \( b[1] \) | \( b[i] \) | \( b[9] \) |
|--------------|------------|------------|------------|
|              | base       | base + esize | Address of \( b[i] \) | base + 9* esize |

```
Implementation of 2-D Arrays (Logical to Physical Mapping)

```c
int b[2][4];
```

```
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\multicolumn{8}{|c|}{b[0][0]} \\
\multicolumn{8}{|c|}{b[0][1]} \\
\multicolumn{8}{|c|}{b[0][2]} \\
\multicolumn{8}{|c|}{b[0][3]} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\hline
\multicolumn{8}{|c|}{b[1][0]} \\
\multicolumn{8}{|c|}{b[1][1]} \\
\multicolumn{8}{|c|}{b[1][2]} \\
\multicolumn{8}{|c|}{b[1][3]} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\hline
\multicolumn{8}{|c|}{b[2][0]} \\
\multicolumn{8}{|c|}{b[2][1]} \\
\multicolumn{8}{|c|}{b[2][2]} \\
\multicolumn{8}{|c|}{b[2][3]} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\hline
\end{array}
```
Two Ways to Map 2-D Arrays
(Logical to Physical Mapping)

• **Row Major Indexing**
  First row occupies the first set of memory locations, then the second row and so on. Also known as lexicographic order.

• **Column Major Indexing**
  First column occupies the first set of memory locations, then the second column and so on.
Row Major Indexing

```c
int b[r1][r2];
```

Physical Address of \( b[i][j] = base + [i \times r2 + j] \times esize \)

How about Column Major Indexing??

Generalization to Multidimensional Arrays
Some Higher Level Abstract Data Structures

- Stacks (Chapter 2)
- Queues (Chapter 4)

Usage: to hold ordered lists. Examples:
- Math expression  
  \[ 2 \times 3 + 4 \]
  \[ 4 + 2 \times 3 \]
  Order (priority) of execution of operators
- Time events (arrivals, departures, . . . )
- Values in a deck of cards
What is Stack?

Insertion (push)

Deletion (pop)

Growth
What is Stack?

Last In First Out (LIFO)

(i) Queue
   1) Type of Element
   2) Size

(ii) Operations
   PUSH —> Test Overflow pre-condition
   POP —> Test Underflow pre-condition
Stacks

- **Operations**
  (i) Empty: to check for empty
  (ii) PUSH (+ Test)
  (iii) POP (test for empty)
  (iv) Stack top

What is its value of the top element without popping?
Stacks (cont.)

• Implementation
  - Array
  - Pointers

An Application of Stack:

Prefix, Infix, Postfix Representation of Expressions
Formal Definition

- structure STACK (item)

1 declare CREATE () stack
2 ADD (item, stack) stack
3 DELETE (stack) stack
4 TOP (stack) item
5 ISEMETS (stack) boolean;
6 for all S ∈ stack, i ∈ item let
7 ISEMETS (CREATE) ::= true
8 ISEMETS (ADD (i,S)) ::= false
9 DELETE (CREATE) ::= error
10 DELETE (ADD (i,S)) ::= S
11 TOP (CREATE) ::= error
12 TOP (ADD (i,S)) ::= I
13 end
14 end STACK
Representing Stacks in C

Implementation using 1-D Array
Both are ordered collection of items

Problems:
1. Stack is dynamic, Array is static
2. Access modes are different

Solutions:
1. Make bounds large
2. Use an index variable to track top of stack (track boundary conditions)
# Implementation and Operations on Stacks

```c
#define STACKSIZE 100
#define INTGR 1
#define FLT 2
#define STRING 3

struct stackelement {
    int etype /* etype equals INTGR, FLT, or STRING */
    /*      depending on the type of the */
    /*           corresponding element.  */

    union {
        int ival;
        float fval;
        char *pval; /* pointer to a string */
    } element;
};

struct stack {
    int top;
    struct stackelement items [STACKTOP];
};
```
Summary of Pros and Cons for Using Linked Lists

Advantages:
Flexibility due to dynamic storage
No overflow problem
Changes (Insertion and Deletions)

Disadvantages:
Space Use
Random Access
Programming Inconvenience
Application: The Polish Notation

Representing Arithmetic Expressions that are easy to process by the computer using some simple data structure such as stack.

- Prefix, Infix, Postfix Representation of Expressions
- Evaluation of Expressions
- Conversion (Bottom-up Parsing)
Priority of Operations

To understand an infix expressions meaning, we need to know the priority of operations!

\[ 2 + 3 \times 2 \]

Does this mean:

(a) \((2 + 3) \times 2\)

or

(b) \(2 + (3 \times 2)\)

Priorities:
(Perform highest priority operations first)
# Priority of Operations (cont.)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>^ , all unary operators, eg, sign (+,-)</td>
<td>6</td>
</tr>
<tr>
<td>x, /, %</td>
<td></td>
</tr>
<tr>
<td>+   - (binary)</td>
<td></td>
</tr>
<tr>
<td>==   !=</td>
<td></td>
</tr>
<tr>
<td>&lt;   &lt;=   &gt;   &gt;=</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>


Left Versus Right Association

A + B + C is understood as (A + B) + C

Left —> Right

If we indicate exponentiation as ^, then for:

A ^ B ^ C

we assume:

A^(B ^ C)

2^2^3 = 2^8
Polish Expressions: Postfix, Prefix and Infix

**Postfix** (Reverse Polish Notation or Suffix Form):
All operators appear after their operand(s):
AB+  (HP calculators)

**Prefix:**
Every operator is written before its operand(s):
+AB

**Infix:**
Customary way
A+B

**Example:**

<table>
<thead>
<tr>
<th>Infix</th>
<th>Postfix</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B &lt; C</td>
<td>A B + C&lt;</td>
<td>&lt;+A BC</td>
</tr>
</tbody>
</table>
Advantages of Postfix:

- No more parenthesizing
- Priority of each operator is no more relevant (easy to implement in the computer)
Evaluation of Postfix Expression

Using a Stack (can be implemented recursively)

Scan the Postfix expression from Left to Right.

While there are tokens (symbols) in the expression:
* Get the next token.
* If token is an operand, push it on the stack.
* Otherwise, remove from the stack, the correct number of operands for the operator, evaluate and push the result back on the stack.
Example
/* EvaluatePostfix: evaluate expression in postfix form. */
Value_type EvaluatePostfix(void)
{
    Kind_type type;
    Token_type token;
    Value_type x, y;
    Stack_type stack;

    Initialize(&stack);
    do {
        getToken(token);
        switch (type = Kind(token)) {
        case OPERAND:
            Push(GetValue(token), &stack);
            break;
        case UNARYOP:
            Pop(&x, &stack);
            Push(DoUnary(token, x), &stack);
            break;
        case BINARYOP:
            Pop(&y, &stack);
            Pop(&x, &stack);
            Push(DoBinary(token, x, y), &stack);
            break;
        case ENDEXPR:
            Pop(&x, &stack);
            if (!Empty(&stack))
                Error("Incorrect expression");
            break;
        }
    } while (type != ENDEXPR);
    return x;
}
Conversion from Infix to Postfix

1. Fully parenthesize the infix expression by taking precedence into account.
2. Move all operators so that they replace their corresponding right parenthesis.
3. Delete all parentheses.

\[ A + B + C \implies AB+C+ \]
\[ A^B^C \implies ABC^^ \]
Conversion Algorithm: Infix to Postfix (cont’d)

Convert the parenthesized expression first (recursively), that is, within a parenthesis perform this step. Operations with the highest precedence are converted first. Once an operation is converted, treat the operator and its operands as a single operand. Take into account the associativity rule.
Example
Algorithm for Infix $\rightarrow$ Prefix

This algorithm is similar to infix to postfix, except operators come before the operand.

Ex: $A^B*C-D+E/F/(G+H)$
Implementation Infix —> Postfix:

Problem with this algorithm: requires two passes.

But, we have learned that the order of the operands is unaffected by the conversion process!!

- We can scan an expression for the first time, passing operands to output.
- Store operators on stack until the correct moment to output.
<table>
<thead>
<tr>
<th>Level</th>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>17R</td>
<td>::</td>
<td>global scope (unary)</td>
</tr>
<tr>
<td>17L</td>
<td>::</td>
<td>class scope (binary)</td>
</tr>
<tr>
<td>16L</td>
<td>-&gt;, .</td>
<td>member selectors</td>
</tr>
<tr>
<td>16L</td>
<td>[ ]</td>
<td>array index</td>
</tr>
<tr>
<td>16L</td>
<td>()</td>
<td>function call</td>
</tr>
<tr>
<td>16L</td>
<td>( )</td>
<td>type construction</td>
</tr>
<tr>
<td>15R</td>
<td>sizeof</td>
<td>size in bytes</td>
</tr>
<tr>
<td>15R</td>
<td>++, --</td>
<td>increment, decrement</td>
</tr>
<tr>
<td>15R</td>
<td>-</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>15R</td>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>15R</td>
<td>+,-</td>
<td>unary minus, plus</td>
</tr>
<tr>
<td>15R</td>
<td>*, &amp;</td>
<td>dereference, address-of</td>
</tr>
<tr>
<td>15R</td>
<td>( )</td>
<td>type conversion (cast)</td>
</tr>
<tr>
<td>15R</td>
<td>new, delete</td>
<td>free store management</td>
</tr>
<tr>
<td>14L</td>
<td>-*,. *</td>
<td>member pointer selectors</td>
</tr>
<tr>
<td>13L</td>
<td>* , / , %</td>
<td>multiplicative operators</td>
</tr>
<tr>
<td>12L</td>
<td>+ , -</td>
<td>arithmetic operators</td>
</tr>
<tr>
<td>11L</td>
<td>&lt;&lt;, &gt;&gt;</td>
<td>bitwise shift</td>
</tr>
<tr>
<td>10L</td>
<td>&lt; , &lt;= , &gt;=</td>
<td>relational operators</td>
</tr>
<tr>
<td>9L</td>
<td>== , !=</td>
<td>equality, inequality</td>
</tr>
<tr>
<td>8L</td>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>7L</td>
<td>-</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>6L</td>
<td>!</td>
<td>bitwise OR</td>
</tr>
<tr>
<td>5L</td>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td>4L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3L</td>
<td>?:</td>
<td>arithmetic if</td>
</tr>
<tr>
<td>2R</td>
<td>= , *= , /=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>2R</td>
<td>%= , += , -= , &lt;&lt;=</td>
<td></td>
</tr>
<tr>
<td>2R</td>
<td>&gt;&gt;= , &amp;= ,</td>
<td>= , ^=</td>
</tr>
<tr>
<td>1L</td>
<td>,</td>
<td>comma operator</td>
</tr>
</tbody>
</table>

Table 2.4 Operator Precedence and Associativity
A Priority Based Algorithm Using Stack

Two priorities for operators:

*in-stack-priority (isp)*

*in-coming-priority (icp)*

For all operators:  \( icp = isp = \) given in the table except for (  

\( isp \) of (  is the lowest = 0

\( icp \) of (  is the highest = 20
1. Pass operands immediately to output (do not push on the stack).

2. Operators in the stack are taken out to the output expression as long as their \textit{isp}'s are greater than or equal to the \textit{icp} of incoming operator. After popping out these operators, the incoming operator is pushed on the stack.

3. If incoming token is \textit{)}, keep on popping the operators until a \textit{(} is encountered to eliminate the pair of parentheses.
Example 1: \( A \times (B + C) \times D \)

<table>
<thead>
<tr>
<th>Next Token</th>
<th>Stack</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>empty</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>empty</td>
<td>A</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>A</td>
</tr>
<tr>
<td>(</td>
<td>x (</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>x (</td>
<td>AB</td>
</tr>
<tr>
<td>+</td>
<td>x ( +</td>
<td>AB</td>
</tr>
<tr>
<td>C</td>
<td>x ( +</td>
<td>ABC</td>
</tr>
<tr>
<td>)</td>
<td>x</td>
<td>ABC+</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>ABC+x</td>
</tr>
<tr>
<td>D</td>
<td>x</td>
<td>ABC+xD</td>
</tr>
</tbody>
</table>

Since there is no more expression, pop all the operators giving:

A B C + x D x